

# Step 1 - Why Docker?

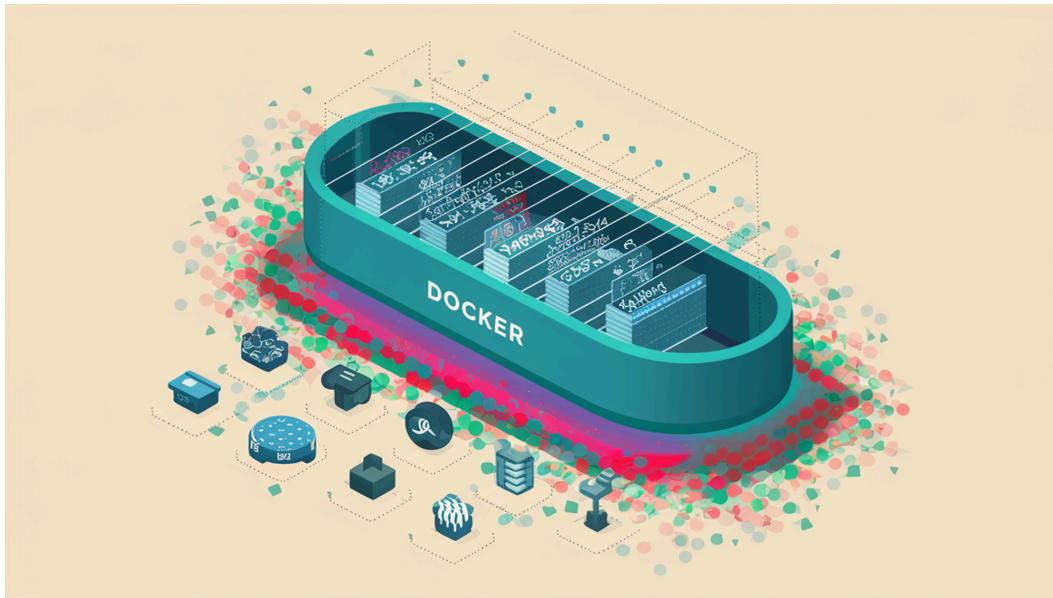


Docker/containers are important for a few reasons -

1. Kubernetes/Container orchestration
2. Running processes in isolated environments
3. Starting projects/auxiliary services locally

# Step 2 - Containerization

## What are containers



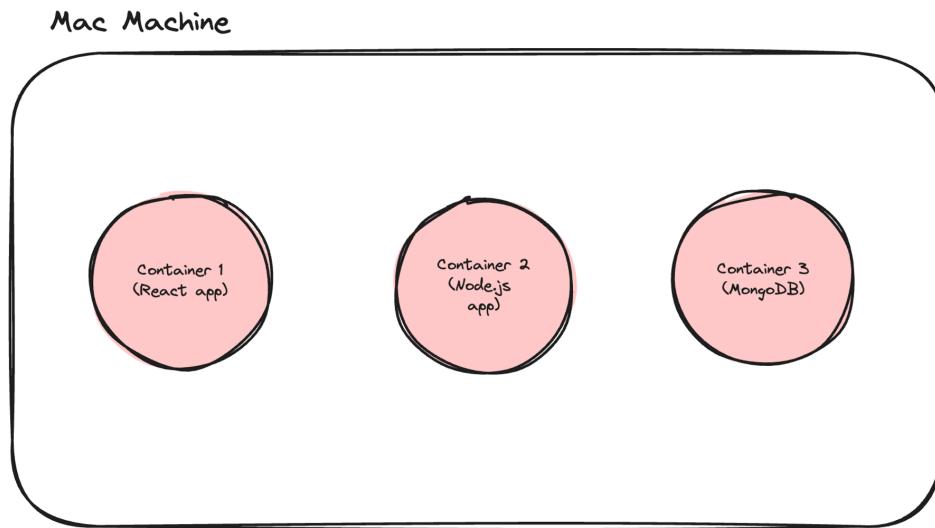
Containers are a way to package and distribute software applications in a way that makes them easy to deploy and run consistently across different environments. They allow you to package an application, along with all its dependencies and libraries, into a single unit that can be run on any machine with a container runtime, such as Docker.

## Why containers



1. Everyone has different Operating systems
2. Steps to run a project can vary based on OS
3. Extremely harder to keep track of dependencies as project grows

## Benefits of using containers



1. Let you describe your **configuration** in a single file
2. Can run in isolated environments
3. Makes Local setup of OS projects a breeze
4. Makes installing auxiliary services/DBs easy

## References

- For Reference, the following command starts **mongo** in all operating systems -

```
docker run -d -p 27017:27017 mongo
```



- Docker isn't the only way to create containers

# Step 3 - History of Docker



Docker is a YC backed company, started in ~2014

They envisioned a world where containers would become mainstream and people would deploy their applications using them

That is mostly true today

Most projects that you open on Github will/should have docker files in them (a way to create docker containers)

Ref - <https://www.ycombinator.com/blog/solomon-hykes-docker-dotcloud-interview/>

# Step 4 - Installing docker

<https://docs.docker.com/engine/install/>

The screenshot shows the Docker Engine installation documentation. At the top, there's a navigation bar with links to 'Manuals', 'Docker Engine', 'Install', and 'Overview'. Below that is a large heading 'Install Docker Engine'. A text block explains that this section covers installing Docker Engine on Linux (Docker CE), and it also notes that Docker Desktop is available for Windows, macOS, and Linux. Below this, a bulleted list provides links to Docker Desktop for Linux, Mac (macOS), and Windows.

Platform	x86_64 / amd64	arm64 / aarch64	arm (32-bit)	ppc64le	s390x
<a href="#">CentOS</a>	✓	✓		✓	
<a href="#">Debian</a>	✓	✓	✓	✓	
<a href="#">Fedora</a>	✓	✓		✓	
<a href="#">Raspberry Pi OS (32-bit)</a>			✓		
<a href="#">RHEL (s390x)</a>				✓	
<a href="#">SLES</a>				✓	
<a href="#">Ubuntu</a>	✓	✓	✓	✓	✓
<a href="#">Binaries</a>	✓	✓	✓		

Make sure you're able to run the `docker cli` locally -

```
→ ~ docker
```

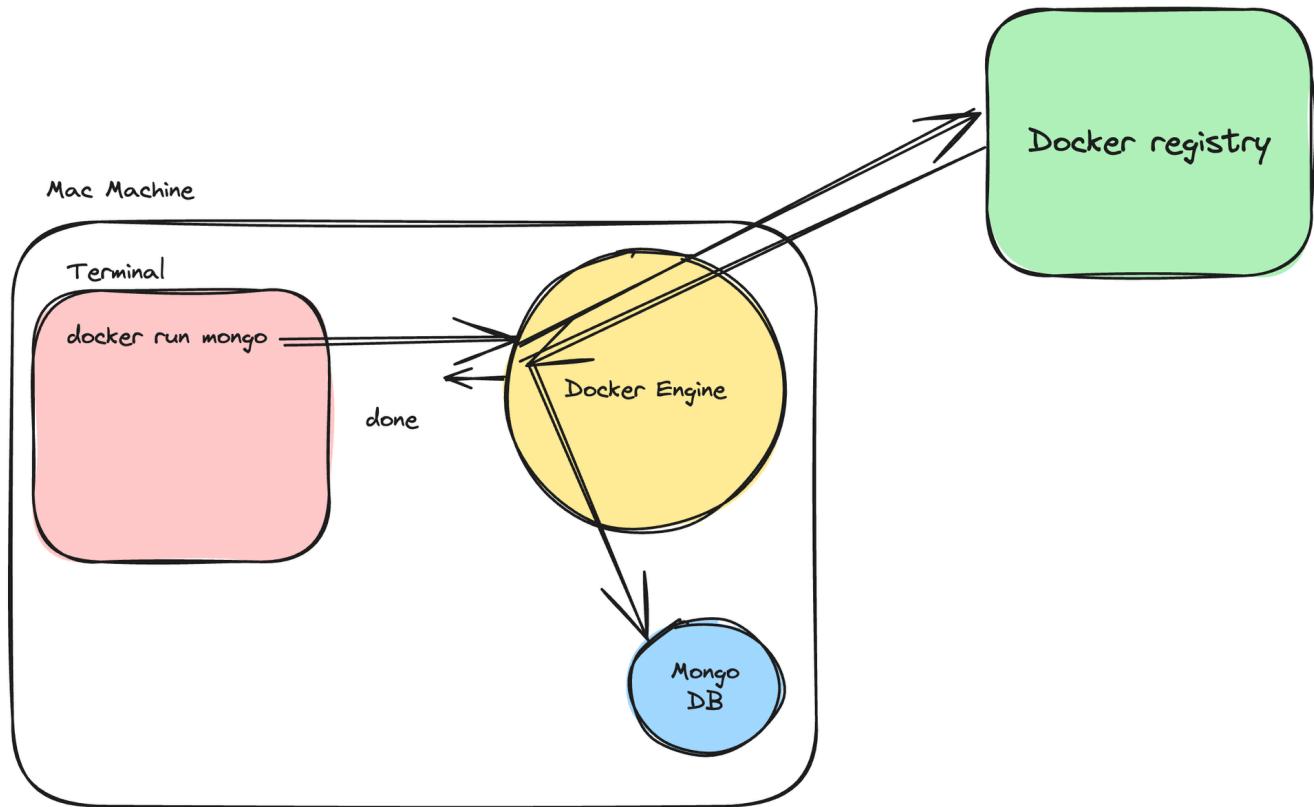
Usage: docker [OPTIONS] COMMAND

A self-sufficient runtime for containers

Common Commands:

run	Create and run a new container from an image
exec	Execute a command in a running container
ps	List containers
build	Build an image from a Dockerfile
pull	Download an image from a registry
push	Upload an image to a registry
images	List images
login	Log in to a registry
logout	Log out from a registry
search	Search Docker Hub for images
version	Show the Docker version information
info	Display system-wide information

# Step 5 - Inside docker



As an application/full stack developer, you need to be comfortable with the following terminologies -

1. Docker Engine
2. Docker CLI - Command line interface
3. Docker registry

## 1. Docker Engine

Docker Engine is an open-source **containerization** technology that allows developers to package applications into **container**

Containers are standardized executable components combining application source code with the operating system (OS) libraries and dependencies required to run that code in any environment.

## 2. Docker CLI

The command line interface lets you talk to the `docker engine` and lets you start/stop/list containers

```
docker run -d -p 27017:27017 mongo
```



Docker cli is not the only way to talk to a docker engine. You can hit the docker `REST API` to do the same things

### 3. Docker registry

The `docker registry` is how Docker makes money.

It is similar to `github`, but it lets you push `images` rather than `sourcecode`

Docker's main registry - <https://dockerhub.com/>

Mongo image on docker registry - [https://hub.docker.com/\\_/mongo](https://hub.docker.com/_/mongo)

# Step 6 - Images vs containers

## Docker Image

A Docker image is a lightweight, standalone, executable package that includes everything needed to run a piece of software, including the code, a runtime, libraries, environment variables, and config files.



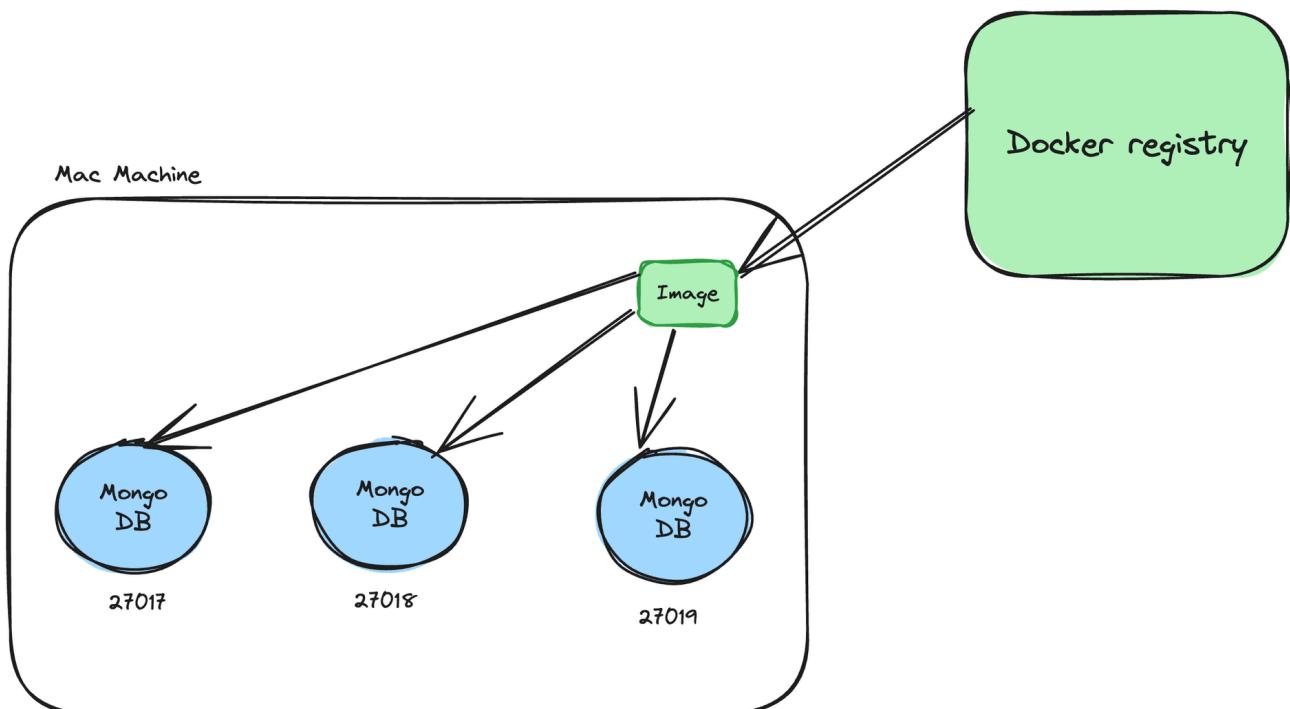
A good mental model for an image is `Your codebase on github`

## Docker Container

A container is a running instance of an image. It encapsulates the application or service and its dependencies, running in an isolated environment.

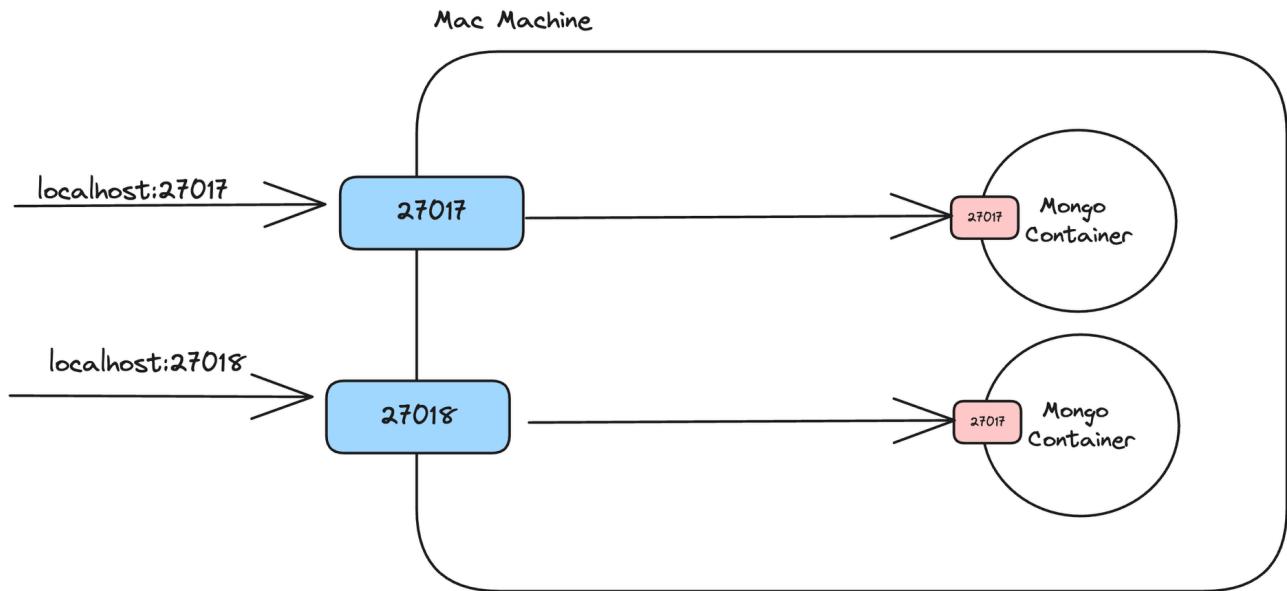


A good mental model for a container is when you run `node index.js` on your machine from some source code you got from github



# Step 7 - Port mapping

```
docker run -d -p 27018:27017 mongo
```



# Step 8 - Common docker commands

1. docker images

2. docker ps

3. docker run

4. docker build

## 1. docker images

Shows you all the images that you have on your machine

## 2. docker ps

Shows you all the containers you are running on your machine

## 3. docker run

Lets you start a container

1. -p ⇒ let's you create a port mapping

2. -d. ⇒ Let's you run it in detached mode

## 4. docker build

Lets you build an image. We will see this after we understand how to create your own

**Dockerfile**

## 5. docker push

Lets you push your image to a registry

## 6. Extra commands

1. docker kill

## 2. docker exec

# Step 9 - Dockerfile

## What is a Dockerfile

If you want to create an image from your own code, that you can push to [dockerhub](#), you need to create a `Dockerfile` for your application.

A Dockerfile is a text document that contains all the commands a user could call on the command line to create an image.

## How to write a dockerfile

A dockerfile has 2 parts

1. Base image
2. Bunch of commands that you run on the base image (to install dependencies like Node.js)

## Let's write our own Dockerfile

Let's try to containerise this backend app - <https://github.com/100xdevs-cohort-2/week-15-live-1>

```
📄 Dockerfile
1  FROM node:16-alpine → Base Image
2
3  WORKDIR /app → Working directory
4
5  COPY . . → Copy over files
6
7  RUN npm install → Run Commands to build the code
8  RUN npm run build
9
10 EXPOSE 3000 → Expose ports
11
12 CMD ["node", "dist/index.js"] → Final command that runs when running the
13                                     container
```

### ▼ Solution

```
FROM node:20
```

```
WORKDIR /app
```



```
COPY . .

RUN npm install
RUN npx prisma generate
RUN npm run build

EXPOSE 3000

CMD ["node", "dist/index.js"]
```

## Common commands

- **WORKDIR** : Sets the working directory for any **RUN** , **CMD** , **ENTRYPOINT** , **COPY** instructions that follow it.
- **RUN** : Executes any commands in a new layer on top of the current image and commits the results.
- **CMD** : Provides defaults for executing a container. There can only be one CMD instruction in a Dockerfile.
- **EXPOSE** : Informs Docker that the container listens on the specified network ports at runtime.
- **ENV** : Sets the environment variable.
- **COPY** : Allow files from the Docker host to be added to the Docker image

<https://github.com/100xdevs-cohort-2/week-15-live-1>

# Step 10 - Building images

Now that you have a dockerfile in your project, try building a `docker image` from it

```
docker build -t image_name .
```



Now if you try to look at your images, you should notice a new image created

```
docker images
```



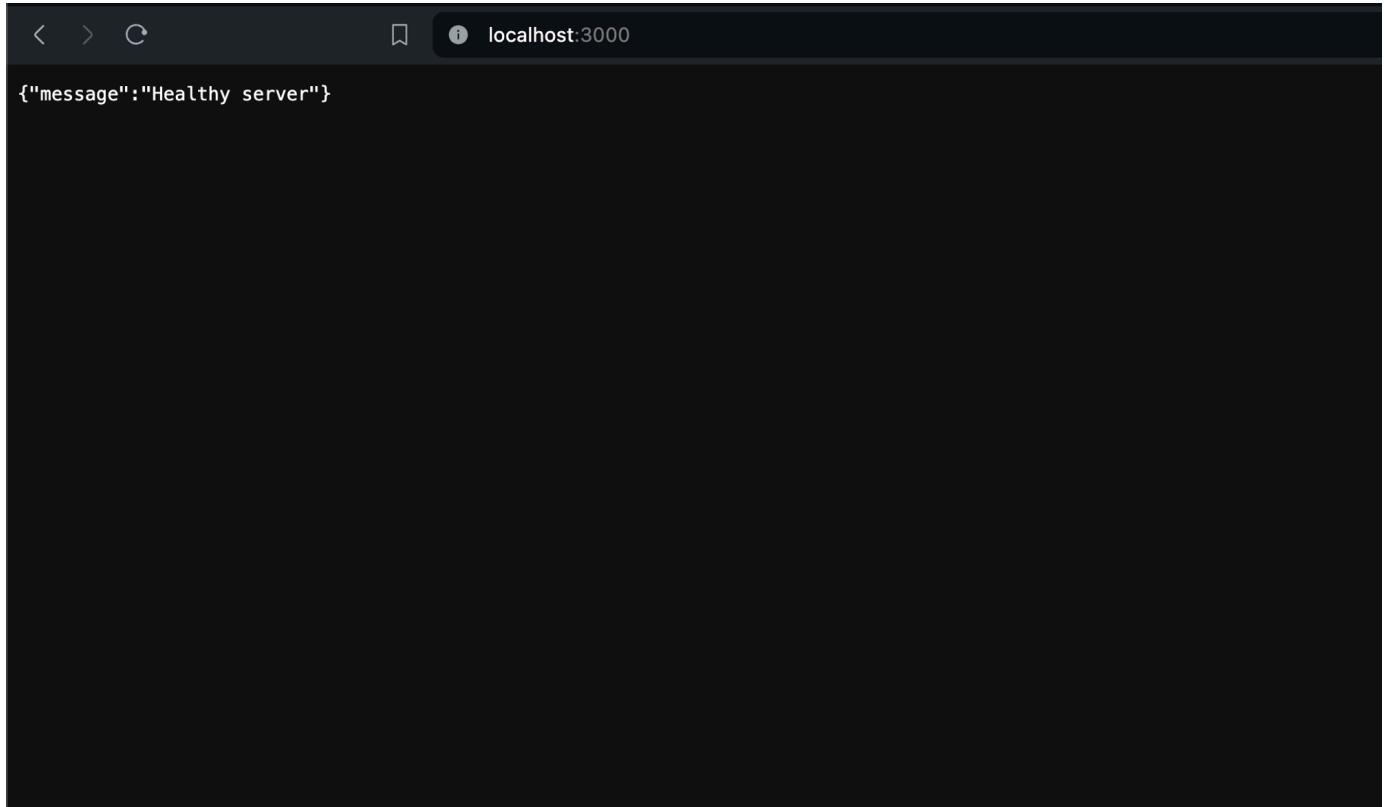
Add a `.dockerignore` so that `node_modules` don't get copied over

# Step 11 - Running images

```
docker run -p 3000:3000 image_name
```



Try visiting [localhost:3000](http://localhost:3000)



# Step 12 - Passing in env variables

```
docker run -p 3000:3000 -e DATABASE_URL="postgres://avnadmin:AVNS_EeDiMIdW-dNT40...
```



The `-e` argument lets you send in environment variables to your node.js app

# Step 13 - More commands

1. docker kill - to kill a container
2. docker exec - to execute a command inside a container

## Examples

1. List all contents of a container folder

```
docker exec <container_name_or_id> ls /path/to/directory
```



1. Running an Interactive Shell

```
docker exec -it <container_name_or_id> /bin/bash
```

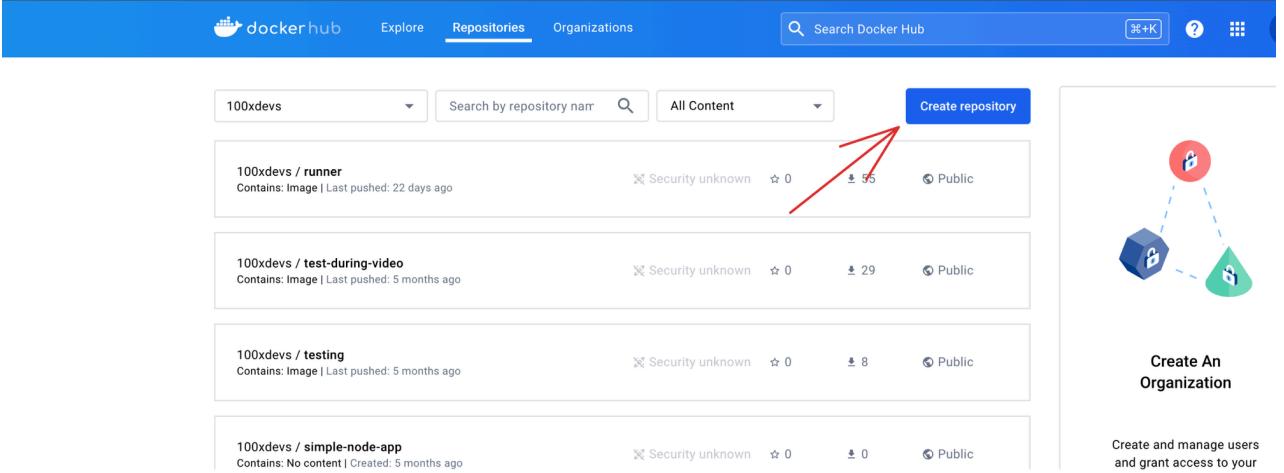


# Step 14 - Pushing to dockerhub

Once you've created your image, you can push it to [dockerhub](#) to share it with the world.

1. Signup to [dockerhub](#)

2. Create a new repository



The screenshot shows the Docker Hub homepage with a blue header. The 'Repositories' tab is selected. Below it, there's a search bar and a 'Create repository' button. The main area displays four repositories under the user '100xdevs': 'runner', 'test-during-video', 'testing', and 'simple-node-app'. Each repository card includes details like 'Contains: Image | Last pushed: 22 days ago' or '5 months ago', security status, star count (0), commit count (55, 29, 8, 0), and a 'Public' badge. To the right of the repositories, there are three icons: a red circle with a white 'F' (Follow), a blue hexagon with a white lock (Security), and a green triangle with a white 'H' (Hub). Below these icons are two buttons: 'Create An Organization' and 'Create and manage users and grant access to your'.

3. Login to docker cli

1. docker login

2. you might have to create an access token - <https://docs.docker.com/security/for-developers/access-tokens/>

4. Push to the repository

```
docker push your_username/your_reponame:tagname
```

# Step 15 - Layers in Docker

In Docker, layers are a fundamental part of the image architecture that allows Docker to be efficient, fast, and portable. A Docker image is essentially built up from a series of layers, each representing a set of differences from the previous layer.



## How layers are made -

- 1. Base Layer:** The starting point of an image, typically an operating system (OS) like Ubuntu, Alpine, or any other base image specified in a Dockerfile.
- 2. Instruction Layers:** Each command in a Dockerfile creates a new layer in the image. These include instructions like `RUN`, `COPY`, which modify the filesystem by installing packages, copying files from the host to the container, or making other changes. Each of these modifications creates a new layer on top of the base layer.
- 3. Reusable & Shareable:** Layers are cached and reusable across different images, which makes building and sharing images more efficient. If multiple images are built from the same base image or share common instructions, they can reuse the same layers, reducing storage space and speeding up image downloads and builds.

**4. Immutable:** Once a layer is created, it cannot be changed. If a change is made, Docker creates a new layer that captures the difference. This immutability is key to Docker's reliability and performance, as unchanged layers can be shared across images and containers.

# Step 16 - Layers practically

For a simple Node.js app - <https://github.com/100xdevs-cohort-2/week-15-live-2>

## Dockerfile

```

Dockerfile
1  FROM node:20
2
3  WORKDIR /usr/src/app
4
5  COPY . .
6
7  RUN npm install
8  RUN npm run build
9  RUN npx prisma generate
10
11
12  EXPOSE 3000
13
14  CMD ["node", "dist/index.js", ]

```

## Logs

```

● → week-15-live-1 git:(main) ✘ docker build -t hkirat-backend-cool-app .
[+] Building 12.8s (12/12) FINISHED                                            docker:desktop-linux
  => [internal] load .dockerignore                                         0.0s
  => => transferring context: 57B                                           0.0s
  => [internal] load build definition from Dockerfile                   0.0s
  => => transferring dockerfile: 189B                                         0.0s
  => [internal] load metadata for docker.io/library/node:20                3.7s
  => [auth] library/node:pull token for registry-1.docker.io               0.0s
  => CACHED [1/6] FROM docker.io/library/node:20@sha256:f3299f16246c71ab8b304d6745bb4059f 0.0s
  => [internal] load build context                                         0.0s
  => => transferring context: 7.55kB                                       0.0s
  => [2/6] WORKDIR /usr/src/app                                         0.0s
  => [3/6] COPY . .                                                 0.0s
  => [4/6] RUN npm install                                              6.0s
  => [5/6] RUN npm run build                                             1.3s
  => [6/6] RUN npx prisma generate                                         1.1s
  => exporting to image                                                 0.5s
  => => exporting layers                                               0.4s
  => => writing image sha256:d0ce15534410858645f5ee075ad09dedebbfafef8353364cf35bf8dafd8 0.0s
  => => naming to docker.io/library/hkirat-backend-cool-app                0.0s

```

## Observations -

1. Base image creates the first layer
2. Each `RUN` , `COPY` , `WORKDIR` command creates a new layer
3. Layers can get re-used across docker builds (notice `CACHED` in 1/6)

# Step 17 - Why layers?

If you change your Dockerfile, layers can get re-used based on where the change was made



If a layer changes, all subsequent layers also change

## Case 1 - You change your source code

```

👉 Dockerfile
1  FROM node:20
2
3  WORKDIR /usr/src/app
4
5  COPY . .
6
7  RUN npm install
8  RUN npm run build
9  RUN npx prisma generate
10
11
12 EXPOSE 3000
13
14 CMD ["node", "dist/index.js", ]

```

Layer 1 - Cached

Layer 2 - Cached

Layer 3 - Changed

Layer 4 - Changed

Layer 5 - Changed

Layer 6 - Changed

## ▼ Logs

```

● → week-15-live-1 git:(main) ✘ docker build -t hkirat-backend-cool-app .
[+] Building 8.7s (11/11) FINISHED
=> [internal] load build definition from Dockerfile
=> => transferring dockerfile: 189B
=> [internal] load .dockerignore
=> => transferring context: 57B
=> [internal] load metadata for docker.io/library/node:20
=> [1/6] FROM docker.io/library/node:20@sha256:f3299f16246c71ab8b304d6745bb4059fa9283e8d025972e28436a9f9b36ed24
=> [internal] load build context
=> => transferring context: 3.51kB
=> CACHED [2/6] WORKDIR /usr/src/app
=> [3/6] COPY . .
=> [4/6] RUN npm install
=> [5/6] RUN npm run build
=> [6/6] RUN npx prisma generate
=> exporting to image
=> => exporting layers
=> => writing image sha256:e0a88f16c2d11e317070c51ec298e61325c0ef4a2214d2c11a2cc063d4b5e338
=> => naming to docker.io/library/hkirat-backend-cool-app

Is still cached, even though it doesn't say it
↑

View build details: docker-desktop://dashboard/build/desktop-linux/desktop-linux/yimvu9plvmbypwatyzd4mdg
What's Next?
View a summary of image vulnerabilities and recommendations → docker scout quickview

```

## Case 2 - You change the package.json file (added a dependency)

Dockerfile

```
1 | FROM node:20                                     Layer 1 - Cached
2 |
3 | WORKDIR /usr/src/app                           Layer 2 - Cached
4 |
5 | COPY . . .                                      Layer 3 - Changed
6 |
7 | RUN npm install                                Layer 4 - Changed
8 | RUN npm run build                             Layer 5 - Changed
9 | RUN npx prisma generate|                      Layer 6 - Changed
10
11
12 EXPOSE 3000
13
14 CMD ["node", "dist/index.js", ]
```

## ▼ Logs

```
● week-15-live-1 git:(main) ✘ docker build -t hkirat-backend-cool-app .
[+] Building 8.7s (11/11) FINISHED
=> [internal] load build definition from Dockerfile
=> => transferring dockerfile: 189B
=> [internal] load .dockerignore
=> => transferring context: 57B
=> [internal] load metadata for docker.io/library/node:20
=> [1/6] FROM docker.io/library/node:20@sha256:f3299f16246c71ab8b304d6745bb4059fa9283e8d025972e28436a9f9b36ed24
=> [internal] load build context
=> => transferring context: 3.51kB
=> CACHED [2/6] WORKDIR /usr/src/app
=> [3/6] COPY . .
=> [4/6] RUN npm install
=> [5/6] RUN npm run build
=> [6/6] RUN npx prisma generate
=> exporting to image
=> => exporting layers
=> => writing image sha256:e0a88f16c2d11e317070c51ec298e61325c0ef4a2214d2c11a2cc063d4b5e338
=> => naming to docker.io/library/hkirat-backend-cool-app

Is still cached, even though it doesn't say it
↑
```

# Thought experiment

How often in a project do you think dependencies change ?

How often does the `npm install` layer need to change?

Wouldn't it be nice if we could `cache` the `npm install` step considering dependencies don't change often?

# Step 18 - Optimising Dockerfile

What if we change the Dockerfile a bit -

```
🚢 Dockerfile
1  FROM node:20
2
3  WORKDIR /usr/src/app
4
5  COPY package* .
6  COPY ./prisma .
7
8  RUN npm install
9  RUN npx prisma generate
10
11 COPY . .
12
13 EXPOSE 3000
14
15 CMD ["node", "dist/index.js", ]
```

## ▼ Dockerfile

```
FROM node:20

WORKDIR /usr/src/app

COPY package* .
COPY ./prisma .

RUN npm install
RUN npx prisma generate

COPY . .

EXPOSE 3000

CMD ["node", "dist/index.js", ]
```



1. We first copy over only the things that `npm install` and `npx prisma generate` need
2. Then we run these scripts
3. Then we copy over the rest of the source code

## Case 1 - You change your source code (but nothing in package.json/prisma)

### Dockerfile

```
1 | FROM node:20                                     Layer 1 - Cached
2 |
3 | WORKDIR /usr/src/app                           Layer 2 - Cached
4 |
5 | COPY package* .                                Layer 3 - Cached
6 | COPY ./prisma .                               Layer 4 - Cached
7 |
8 | RUN npm install                               Layer 5 - Cached
9 | RUN npx prisma generate                     Layer 6 - Cached
10|
11| COPY . .                                     Layer 7 - Uncached
12|
13EXPOSE 3000                                    Layer 8 - Uncached
14|
15CMD ["node", "dist/index.js", ]
```

## Case 2 - You change the package.json file (added a dependency)

 Dockerfile

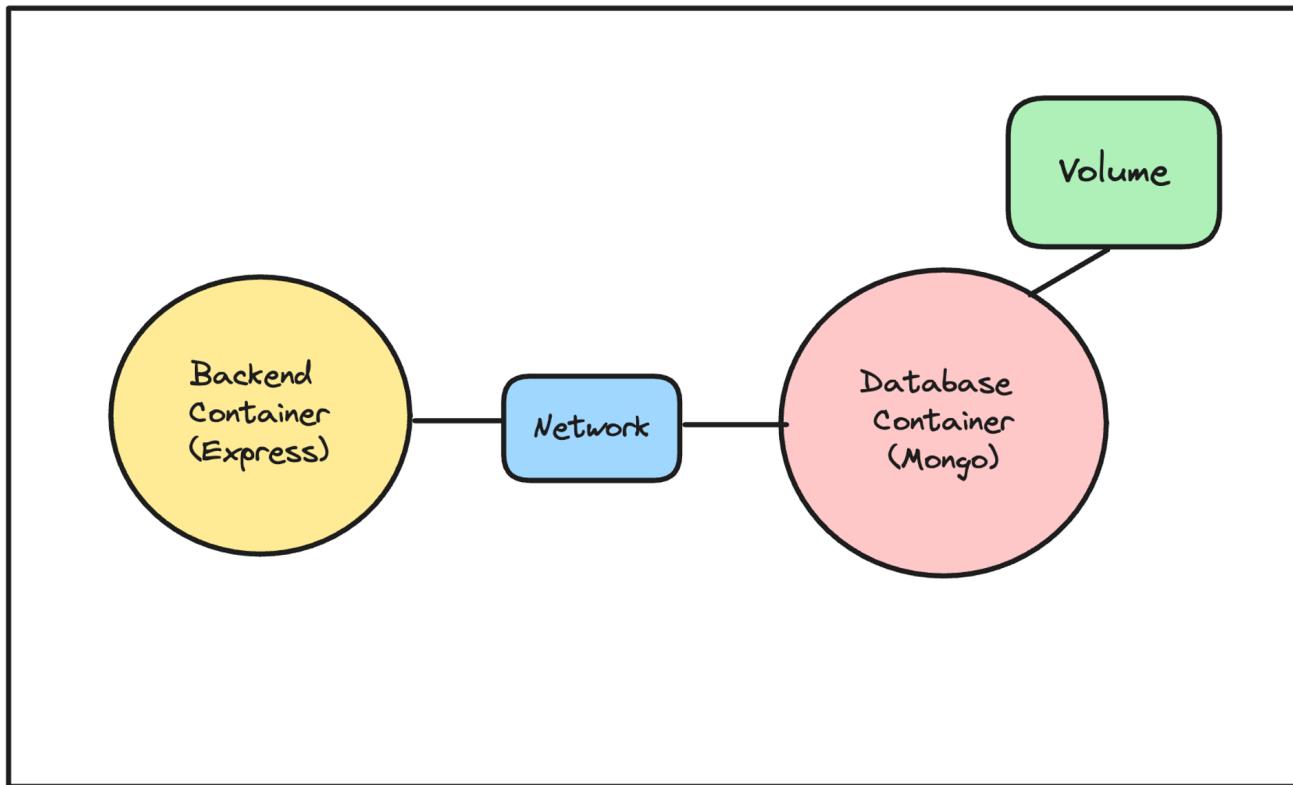
```
1  FROM node:20                                Layer 1 - Cached
2
3  WORKDIR /usr/src/app                         Layer 2 - Cached
4
5  COPY package* .
6  COPY ./prisma .
7
8  RUN npm install                             Layer 5 - Uncached
9  RUN npx prisma generate                    Layer 6 - Uncached
10
11 COPY . .
12
13 EXPOSE 3000                                Layer 8 - Uncached
14
15 CMD ["node", "dist/index.js", ]
```

# Step 19 - Networks and volumes

Networks and volumes are concepts that become important when you have multiple containers running in which you

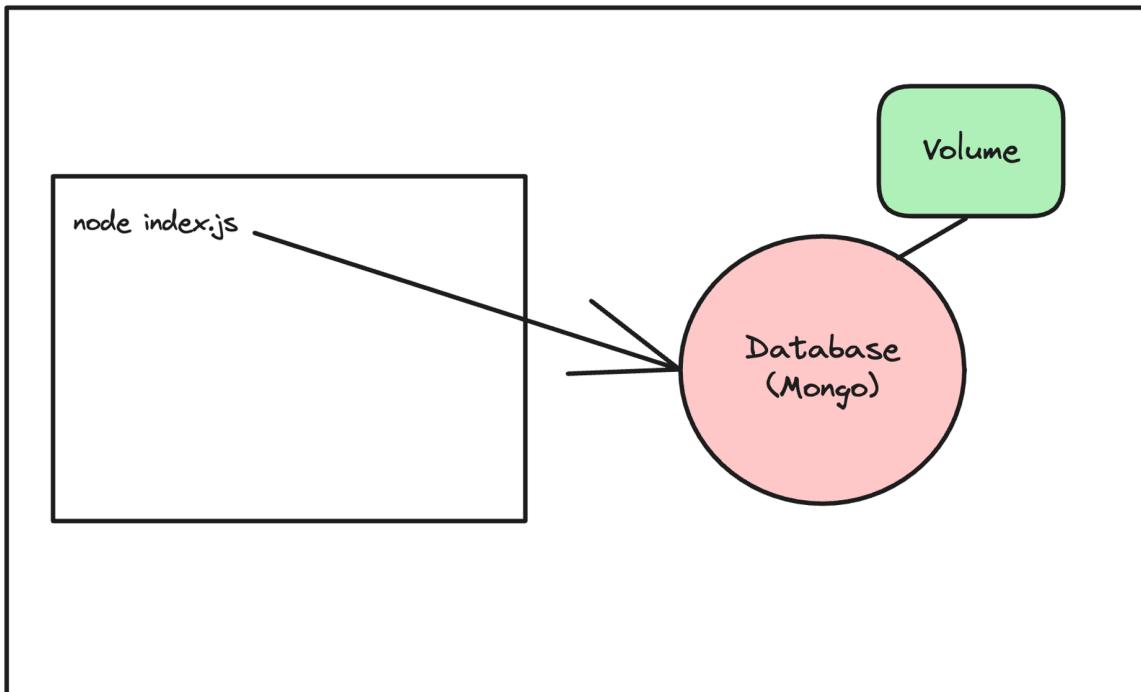
1. Need to persist data across docker restarts
2. Need to allow containers to talk to each other

Mac



We didn't need `networks` until now because when we started the `mongo` container, it was being accessed by a Node.js process running directly on the machine

Mac



# Step 20 - Volumes

If you restart a `mongo` docker container, you will notice that your data goes away.

This is because docker containers are **transitory** (they don't retain data across restarts)

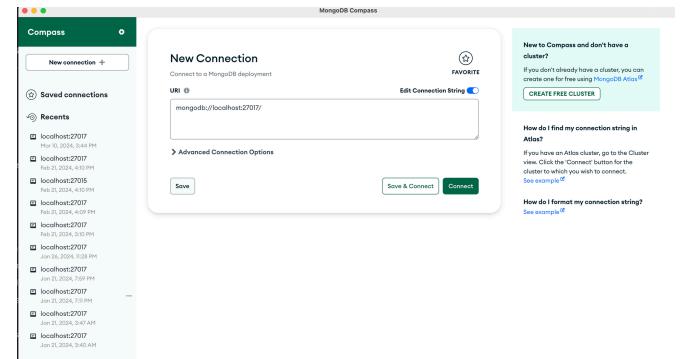
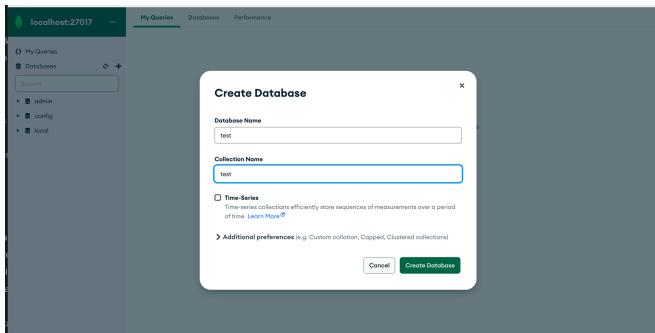
## Without volumes

1. Start a mongo container locally

```
docker run -p 27017:27017 -d mongo
```



1. Open it in MongoDB Compass and add some data to it



1. Kill the container

```
docker kill <container_id>
```



1. Restart the container

```
docker run -p 27017:27017 -d mongo
```



1. Try to explore the database in Compass and check if the data has persisted (it wouldn't)

# With volumes

## 1. Create a `volume`

```
docker volume create volume_database
```



## 1. Mount the folder in `mongo` which actually stores the data to this volume

```
docker run -v volume_database:/data/db -p 27017:27017 mongo
```



## 1. Open it in MongoDB Compass and add some data to it

## 1. Kill the container

```
docker kill <container_id>
```



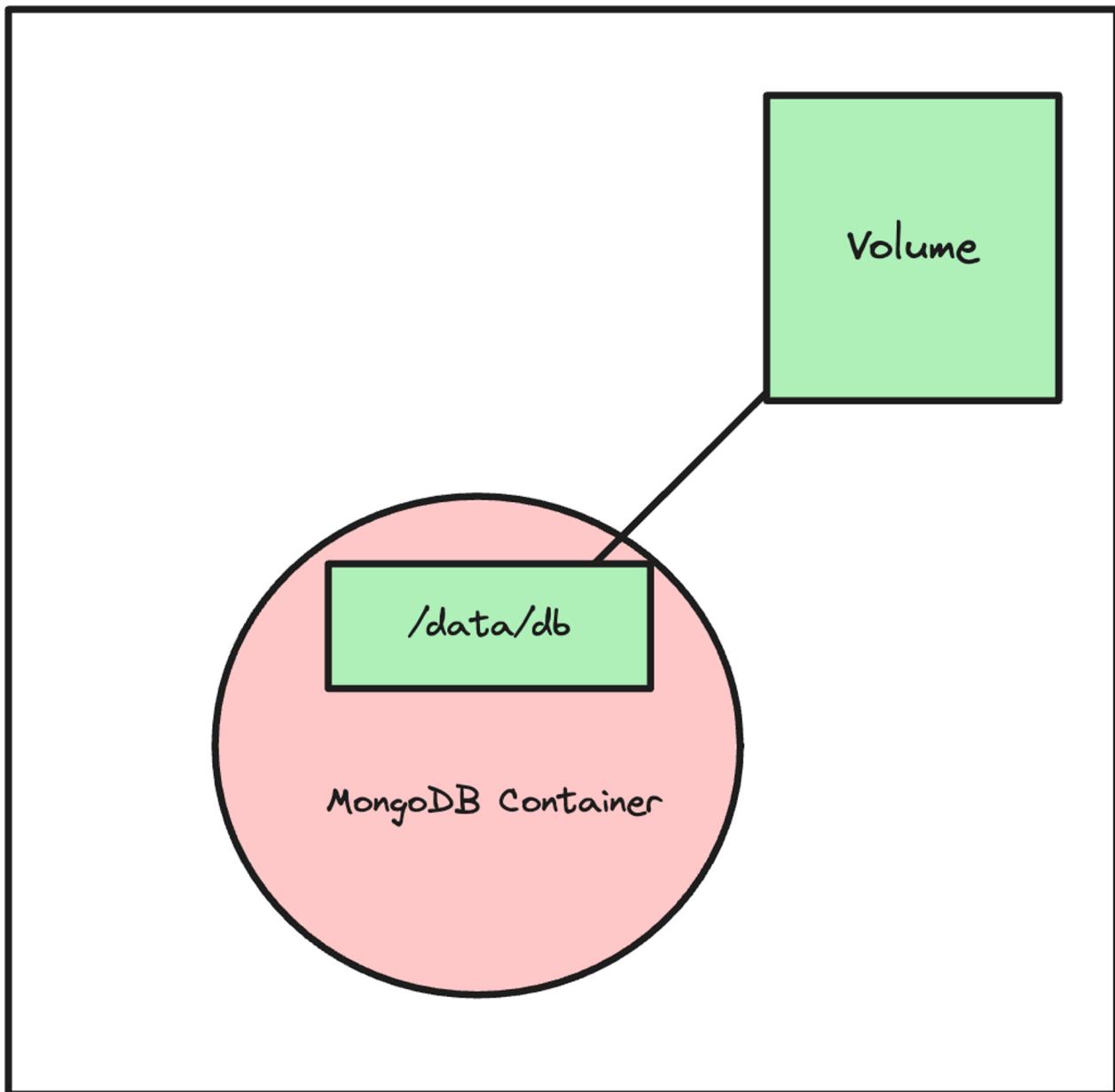
## 1. Restart the container

```
docker run -v volume_database:/data/db -p 27017:27017 mongo
```



## 1. Try to explore the database in Compass and check if the data has persisted (it will!)

Mac



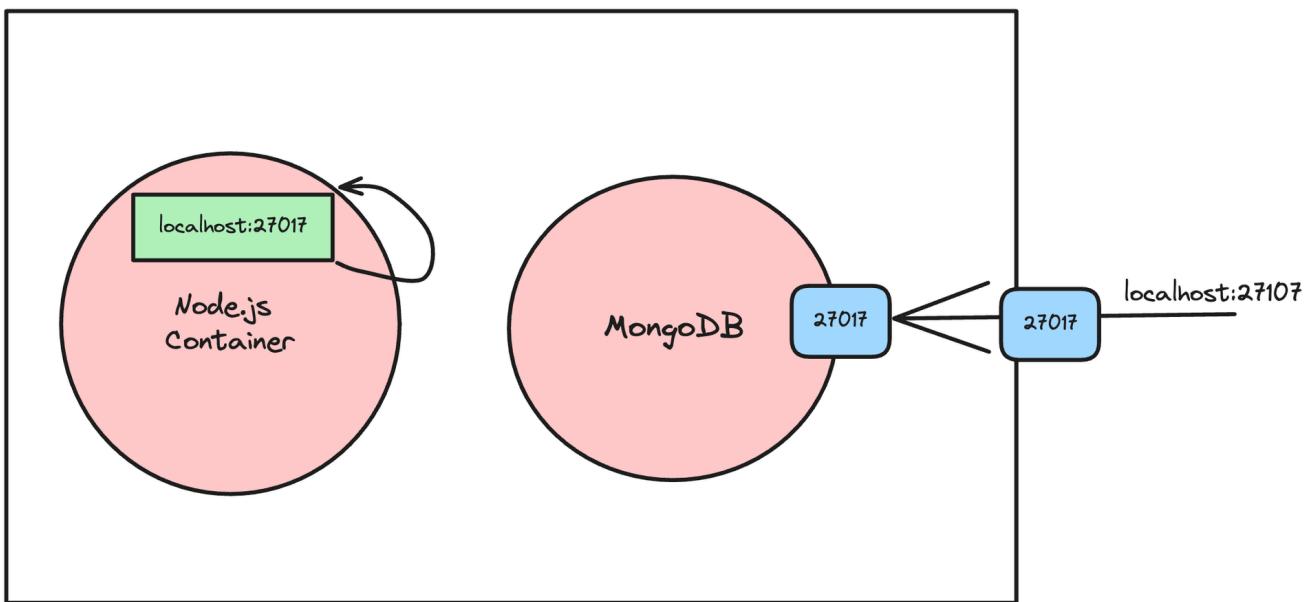
# Step 21 - Network

In Docker, a network is a powerful feature that allows containers to communicate with each other and with the outside world.

Docker containers can't talk to each other by default.

`localhost` on a docker container means `it's own network` and not the network of the `host machine`

Mac Machine



## How to make containers talk to each other?

Attach them to the same network

1. Clone the repo - <https://github.com/100xdevs-cohort-2/week-15-live-2.2>
2. Build the image

```
docker build -t image_tag .
```



1. Create a network

```
docker network create my_custom_network
```



1. Start the `backend process` with the `network` attached to it

```
docker run -d -p 3000:3000 --name backend --network my_custom_network image_tag
```



1. Start mongo on the same network

```
docker run -d -v volume_database:/data/db --name mongo --network my_custom_netwo
```

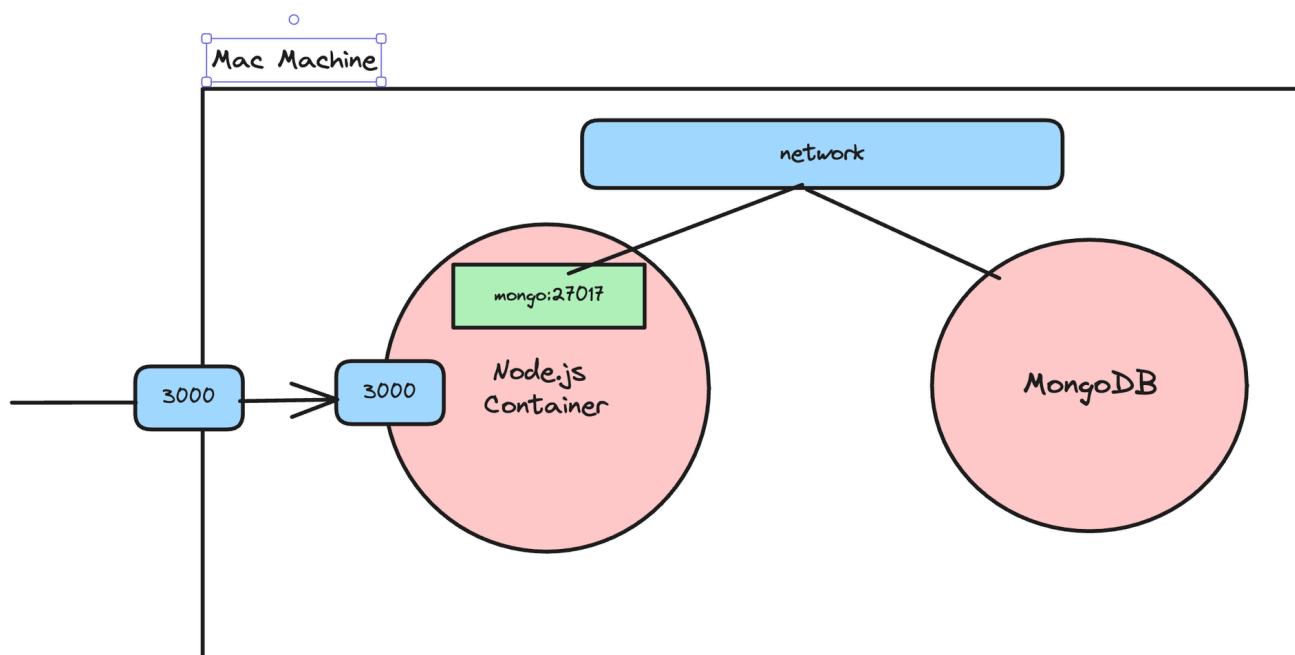


1. Check the logs to ensure the db connection is successful

```
docker logs <container_id>
```



1. Try to visit an endpoint and ensure you are able to talk to the database
2. If you want, you can remove the port mapping for mongo since you don't necessarily need it exposed on your machine



## Types of networks

- **Bridge:** The default network driver for containers. When you run a container without specifying a network, it's attached to a bridge network. It provides a private internal network on the host machine, and containers on the same bridge network can communicate with each other.
- **Host:** Removes network isolation between the container and the Docker host, and uses the host's networking directly. This is useful for services that need to handle lots of traffic or need to expose many ports.

# Step 22 - docker-compose

Docker Compose is a tool designed to help you define and run multi-container Docker applications. With Compose, you use a YAML file to configure your application's services, networks, and volumes. Then, with a single command, you can create and start all the services from your configuration.

```
# docker-compose.yaml
1  version: '3.8'                                Version of docker-compose
2  services:
3    mongodb:
4      image: mongo
5      container_name: mongodb
6      ports:
7        - "27017:27017"
8      volumes:
9        - mongodb_data:/data/db
10
11   backend22:
12     image: backend22
13     container_name: backend_app
14     depends_on:
15       - mongodb
16     ports:
17       - "3000:3000"
18     environment:
19       MONGO_URL: "mongodb://mongodb:27017"
20
21   volumes:
22     mongodb_data:                                Volumes to create
23
```

## Before docker-compose

- Create a network

```
docker network create my_custom_network
```



- Create a volume

```
docker volume create volume_database
```



- Start mongo container

```
docker run -d -v volume_database:/data/db --name mongo --network my_custom_netwo
```



- Start backend container

```
docker run -d -p 3000:3000 --name backend --network my_custom_network backend
```



## After docker-compose

1. Install docker-compose - <https://docs.docker.com/compose/install/>
2. Create a `.yaml` file describing all your containers and volumes (by default all containers in a docker-compose run on the same network)

### ▼ Solution

```
version: '3.8'  
services:  
  mongodb:  
    image: mongo  
    container_name: mongodb  
    ports:  
      - "27017:27017"  
    volumes:  
      - mongodb_data:/data/db
```



```
  backend22:  
    image: backend  
    container_name: backend_app  
    depends_on:  
      - mongodb  
    ports:  
      - "3000:3000"  
    environment:
```

```
MONGO_URL: "mongodb://mongodb:27017"
```

```
volumes:
```

```
  mongodb_data:
```

---

## 1. Start the compose

```
docker-compose up
```



---

## 1. Stop everything (including volumes)

```
docker-compose down --volumes
```



